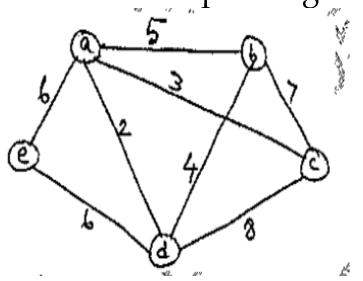
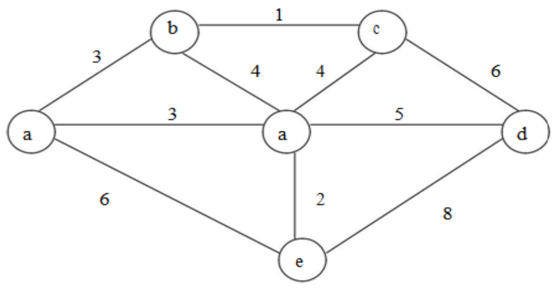
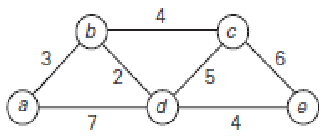
CMR
INSTITUTE OF
TECHNOLOGY

USN	1	C								

Internal Assessment Test II – Nov 2020

Sub:	Design and Analysis of Algorithms							Code:	18MCA33
Date:	04-11-20	Duration:	90 mins	Max Marks:	50	Sem:	III	Branch:	MCA

Note: Answer any full 5 questions. All questions carry equal marks. Total marks: 50

	Marks	OBE CO	OBE RBT
1. Explain and design Prim's algorithm and apply it for the given graph to find minimum cost spanning tree. 	10	CO2 CO4 CO6	L2 L4
2. Find the minimum cost spanning tree for the given graph below by applying Kruskal's algorithm. Write the algorithm and compute minimum cost 	10	CO2 CO4 CO6	L2 L4
3. Apply the Dijkstra's single source shortest path algorithms and analyze its time complexity. Source vertex 'a'. 	10	CO2 CO4 CO6	L2 L4
4. Construct Huffman Tree for the following data. Encode DAD and Decode 100110111110 Symbol : A B C D - Frequency : 0.35 0.1 0.2 0.2 0.15	10	CO2 CO4 CO6	L4
5. Implement Knapsack algorithm on the following data. Maximum capacity of the sack is 8.	10	CO2 CO4 CO6	L4
6. Design an algorithm for string matching problem using brute force technique. Apply it to search a pattern ABABC in the text BAABABABCCA.	10	CO2 CO4 CO6	L2 L4

Item	1	2	3	4
Value	1	2	5	6
Weight	2	3	4	5

7. Discuss Divide and Conquer strategy for designing algorithms. Apply it for multiplication of large integers.	10	CO2 CO4 CO6	L2 L3
8. Write pseudo-code for merge sort and calculate the time complexity.	10	CO2 CO3 CO4 CO6	L2

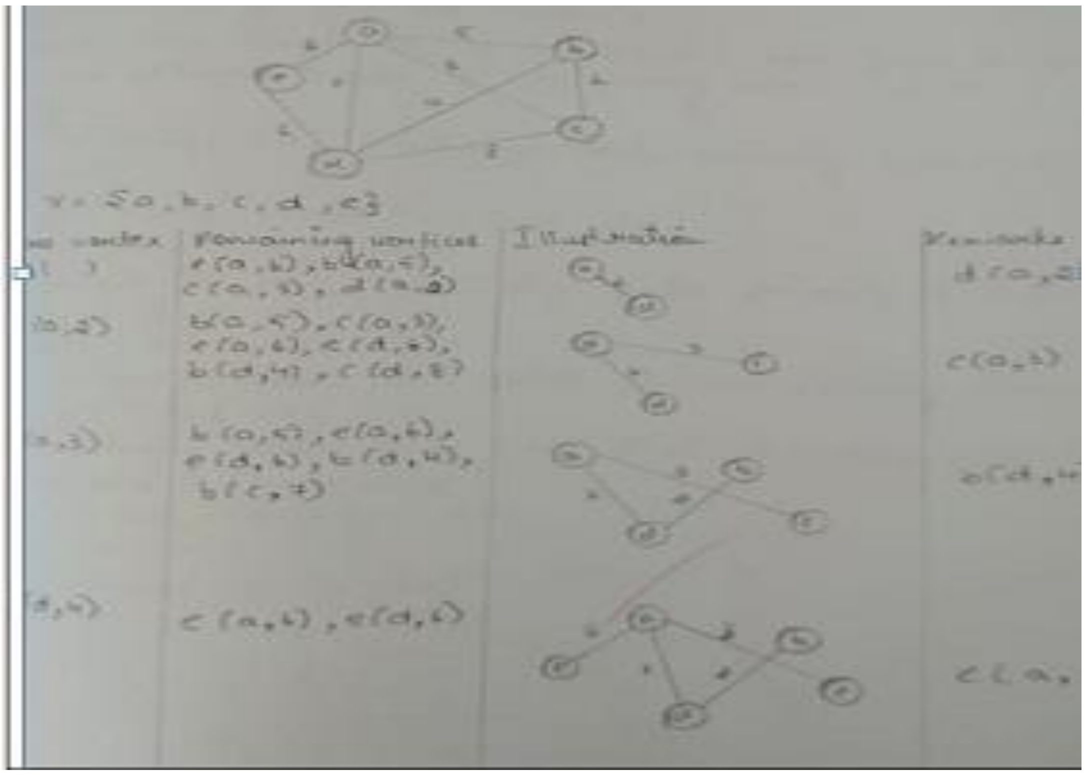
Answers:

1. Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration, the algorithm expands the current tree by simply attaching to it the nearest vertex not in that tree. The algorithm stops after all the graph's vertices have been included in the tree being constructed.

Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is n − 1, where n is the number of vertices in the graph. The tree generated by the algorithm is obtained as the set of edges used for the tree expansions.
Here is pseudocode of this algorithm

```
ALGORITHM  Prim(G)
    //Prim's algorithm for constructing a minimum spanning tree
    //Input: A weighted connected graph $G = \langle V, E \rangle$
    //Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
    $V_T \leftarrow \{v_0\}$    //the set of tree vertices can be initialized with any vertex
    $E_T \leftarrow \varnothing$
    for $i \leftarrow 1$ to $|V| - 1$ do
        find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges $(v, u)$
        such that $v$ is in $V_T$ and $u$ is in $V - V_T$
        $V_T \leftarrow V_T \cup \{u^*\}$
        $E_T \leftarrow E_T \cup \{e^*\}$
    return $E_T$
```

.



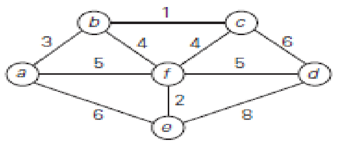
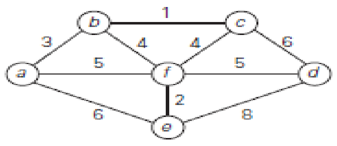
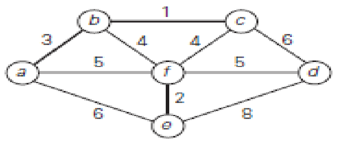
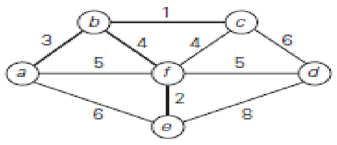
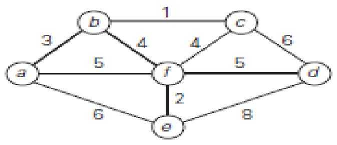
V = {a, b, c, d, e}

Tree vertex	Remaining vertices	Illustration	Remarks
a( )	e(a,6), b(a,5), c(a,3), d(a,2)		d(a,2
d(a,2)	b(a,5), c(a,3), e(a,6), e(d,6), b(d,4), c(d,8)		c(a,3)
c(a,3)	b(a,5), e(a,6), e(d,6), b(d,4), b(c,7)		b(d,4
b(d,4)	e(a,6), e(d,6)		e(a,

Min cost = 2+3+4+6 = 15

2. Kruskal's algorithm is used for solving the minimal spanning tree problem. ***Spanning tree*** of an undirected connected graph is its connected acyclic subgraph(tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.

Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph $G = (V, E)$ as an acyclic subgraph with $|V| - 1$ edges for which the sum of the edge weights is the smallest. Consequently, the algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but are not necessarily connected on the intermediate stages

Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		

```
ALGORITHM  Kruskal(G)
    //Kruskal's algorithm for constructing a minimum spanning tree
    //Input: A weighted connected graph $G = \langle V, E \rangle$
    //Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
    sort $E$ in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
    $E_T \leftarrow \varnothing$;   $ecounter \leftarrow 0$      //initialize the set of tree edges and its size
    $k \leftarrow 0$                          //initialize the number of processed edges
    while $ecounter < |V| - 1$ do
        $k \leftarrow k + 1$
        if $E_T \cup \{e_{i_k}\}$ is acyclic
            $E_T \leftarrow E_T \cup \{e_{i_k}\}$;   $ecounter \leftarrow ecounter + 1$
    return $E_T$
```

3. Dijkstra's algorithm is an algorithm for solving the single-source shortest-paths problem: for a given vertex called the source in a weighted connected graph with non negative edges, find shortest paths to all its other vertices. Some of the applications of the problem are transportation planning, packet routing in communication networks finding shortest paths in social networks, etc. First, it finds the shortest path from the source. to a vertex nearest to it, then to a second nearest, and so on. In general, before its ith iteration starts, the algorithm has already identified the shortest paths to i − 1 other vertices nearest to the source. These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree Ti of the given graph. The set of vertices adjacent to the vertices in T called "fringe vertices"; are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source. To identify the ith nearest vertex, the algorithm computes, for every fringe vertex u, the sum of the distance to the nearest tree vertex v and the length dv of the shortest path from the source to v and then selects the vertex with the smallest such d value. d indicates the length of the shortest path from the source to that vertex till that point. We also associate a value p with each vertex which indicates the name of the next-to-last vertex on such a path, . After we have identified a vertex u* to be added to the tree, we need to perform two operations.

The psuedocode for Dijkstra's is as given below:

- Move u^* from the fringe to the set of tree vertices.
- For each remaining fringe vertex u that is connected to u^* by an edge of weight $w(u^*, u)$ such that $d_{u^*} + w(u^*, u) < d_u$, update the labels of u by u^* and $d_{u^*} + w(u^*, u)$, respectively.

```
ALGORITHM  Dijkstra(G, s)
    //Dijkstra's algorithm for single-source shortest paths
    //Input: A weighted connected graph $G = \langle V, E \rangle$ with nonnegative weights
    //        and its vertex $s$
    //Output: The length $d_v$ of a shortest path from $s$ to $v$
    //          and its penultimate vertex $p_v$ for every vertex $v$ in $V$
    Initialize(Q)   //initialize priority queue to empty
    for every vertex $v$ in $V$
        $d_v \leftarrow \infty$;   $p_v \leftarrow$ null
        Insert(Q, v, $d_v$)   //initialize vertex priority in the priority queue
    $d_s \leftarrow 0$;   Decrease(Q, s, $d_s$)   //update priority of $s$ with $d_s$
    $V_T \leftarrow \varnothing$
    for $i \leftarrow 0$ to $|V| - 1$ do
        $u^* \leftarrow$ DeleteMin(Q)   //delete the minimum priority element
        $V_T \leftarrow V_T \cup \{u^*\}$
        for every vertex $u$ in $V - V_T$ that is adjacent to $u^*$ do
            if $d_{u^*} + w(u^*, u) < d_u$
                $d_u \leftarrow d_{u^*} + w(u^*, u)$;   $p_u \leftarrow u^*$
                Decrease(Q, u, $d_u$)
```

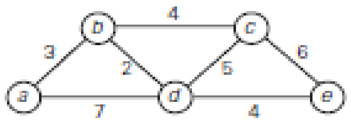
Analysis:

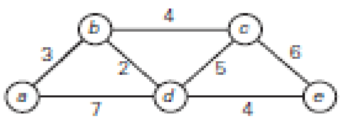
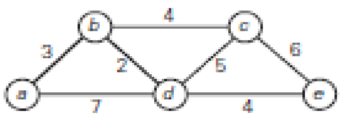
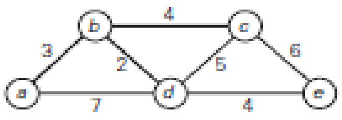
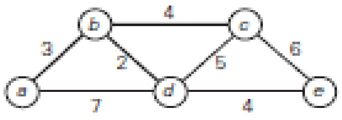
The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself.

Graph represented by adjacency matrix and priority queue by array:

In loop for initialization takes time |V| since the insertion into the queue would just involve appending the vertices at the end(since it is an array implementation). For the second loop, the loop runs |V| times. Each time the DeleteMin operation would take a maximum of θ(|V|) time since it would involve finding the vertex in the array with min d value, for a total time of |V|2. The for loop (for iupdating the neighbor vetices) would

run |V| times again. However the Decrease would take θ(1) time because the index of the vertex would be known.
Thus the total time complexity is θ(|V|2).
Graph represented by adjacency list and priority queue by binary heap:
All heap operations take θ(lg|V|) time. Thus the first loop runs |V| times and each time the Insert would take θ(lg|V|) time. The second loop runs |V| times and the DeleteMin would again take lg|V| time. Thus the total number of time DecreaseMin would run across all iterations is θ(Vlg|V|). In the second loop the basic operation is Decrease(Q,u,du) whoch is run the maximum number of times. Across all iterations using adjacency list, since for each vertex Decrease is called for a maximum of all its adjacent vertices, the number of times Decrease is invoked |E| times. For each time it is onvoked , it takes O(lg|V|) time to execute. Thus the total time complexity is θ((|E|+|V|)lg|V|).
Graph represented by adjacency list and priority queue by fibonacci heap:
The time taken in this case θ(|E|+|V|lg|V|).



Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

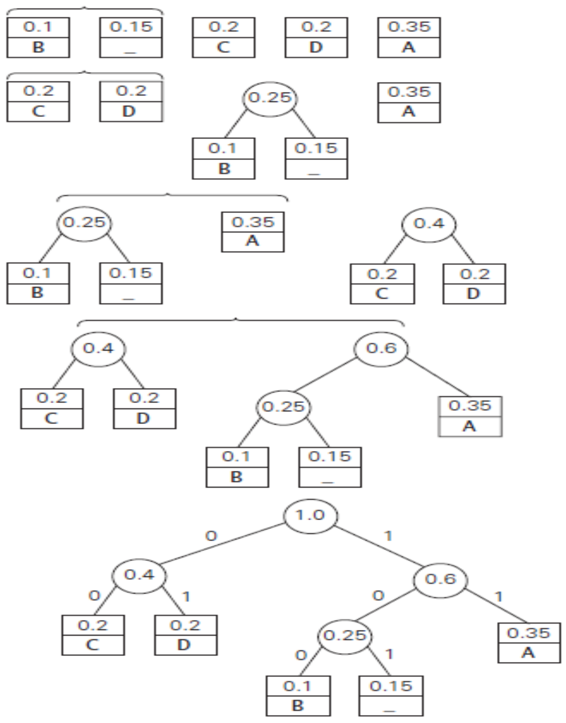
The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

from a to b :	$a - b$	of length 3
from a to d :	$a - b - d$	of length 5
from a to c :	$a - b - c$	of length 7
from a to e :	$a - b - d - e$	of length 9

4. Huffman's algorithm

Step 1 Initialize *n* one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's ***weight***. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)
Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight (ties can be broken arbitrarily, but see Problem 2 in this section's exercises). Make them the left and right

subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.
A tree constructed by the above algorithm is called a ***Huffman tree***. It
defines—in the manner described above—a ***Huffman code***.



The resulting codewords are as follows:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Hence, **DAD** is encoded as **011101**, and
10011011011101 is decoded as **BAD_AD.**

5.

Value	Weight	0	1	2	3	4	5	6	7	8
1	2	0	0	1	1	1	1	1	1	1
2	3	0	0	1	2	2	3	3	3	3
5	4	0	0	1	2	5	5	6	7	7
6	5	0	0	1	2	5	6	6	7	8

6. Given a string of n characters called the text and a string of m characters (m ≤ n) called the pattern, find a substring of the text that matches the pattern. To put it more precisely, we want to find i–the index of the leftmost character of the first matching substring in the text–such that $t_i = p_0, \dots, t_{i+j} = p_j, \dots, t_{i+m-1} = p_{m-1}$:

$$\begin{array}{cccccccccc} t_0 & \dots & t_i & \dots & t_{i+j} & \dots & t_{i+m-1} & \dots & t_{n-1} & \text{text } T \\ & & \updownarrow & & \updownarrow & & \updownarrow & & & \\ & & p_0 & \dots & p_j & \dots & p_{m-1} & & \text{pattern } P & \end{array}$$

If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted. align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all the m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered.

```
Algorithm Brute Force string match (T[0..,n-1], P[0..m-1])
// Input: An array T [0..n-1] of n chars, text
//          An array P [0..m-1] of m chars , a pattern.
// Output: The position of the first character in the text that starts the first
//          matching substring if the search is successful and -1 otherwise.


for i ← 0 to n-m do
        j ← 0
        while j < m and P[j] = T[i+j] do
                j ← j+1
        if j = m return i
return -1
```

Example

Text String = { BAABABABCCA **}**

Pattern String ={ ABABC **}**

B	**A**	**A**	**B**	**A**	**B**	**A**	**B**	**C**	**C**	**A**	
A	**B**	**A**	**B**	**C**							
	A	**B**	**A**	**B**	**C**						
		A	**B**	**A**	**B**	**C**					
			A	**B**	**A**	**B**	**C**				
				A	**B**	**A**	**B**	**C**			

String is matched return the starting Index -4

B	**A**	**A**	**B**	**A**	**B**	**A**	**B**	**C**	**C**	**A**	
				A	**B**	**A**	**B**	**C**			

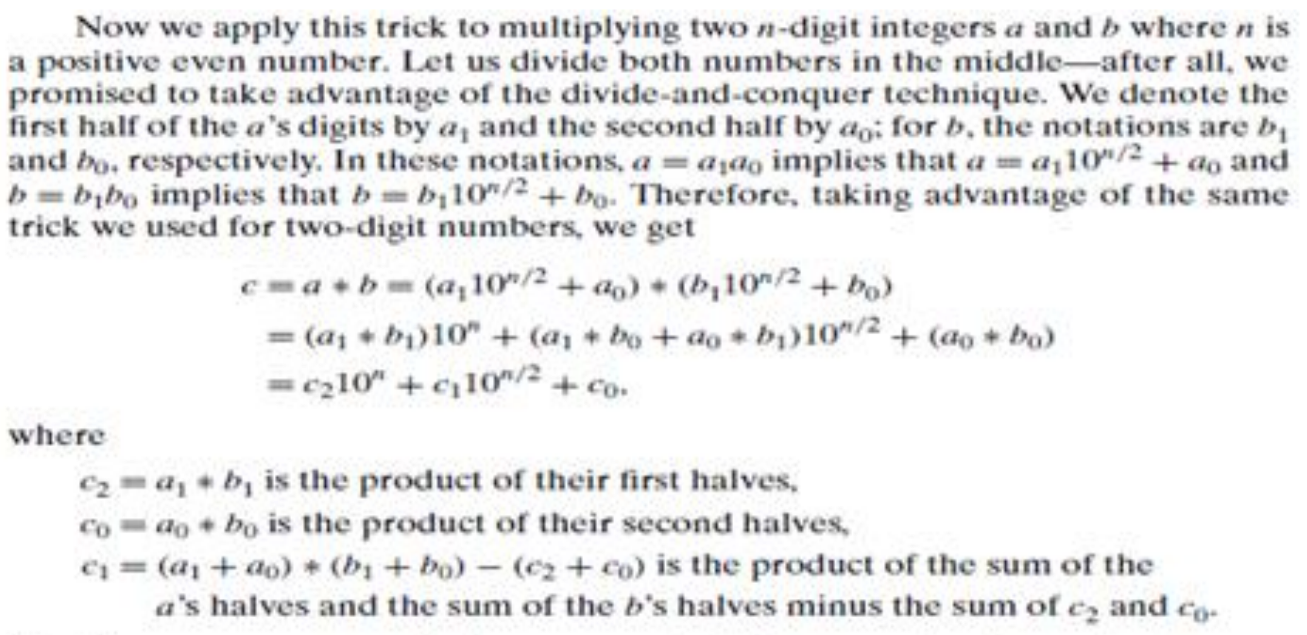
The time complexity would be analyzed by finding the number of times the basic operation j=j+1 is executed.
The inner loop will be executed a maximum of m times (j=0 to m-1).
Therefore

T(n)= $\sum_{i=0}^{n-m}\sum_{j=0}^{m-1} 1 = \sum_{i=0}^{n-m} m$ = (n-m)*m = θ(mn).

Where m is the length of pattern and n is the length of text.

7. The conventional algorithm for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications. (If one of the numbers has fewer digits than the other, we can pad the shorter

number with leading zeros to equalize their lengths.)
By using divide-and-conquer method, it would be possible to design an algorithm with fewer than n^2 digit multiplications,



Now we apply this trick to multiplying two n-digit integers a and b where n is a positive even number. Let us divide both numbers in the middle—after all, we promised to take advantage of the divide-and-conquer technique. We denote the first half of the a's digits by a_1 and the second half by a_0; for b, the notations are b_1 and b_0, respectively. In these notations, $a = a_1a_0$ implies that $a = a_1 10^{n/2} + a_0$ and $b = b_1b_0$ implies that $b = b_1 10^{n/2} + b_0$. Therefore, taking advantage of the same trick we used for two-digit numbers, we get

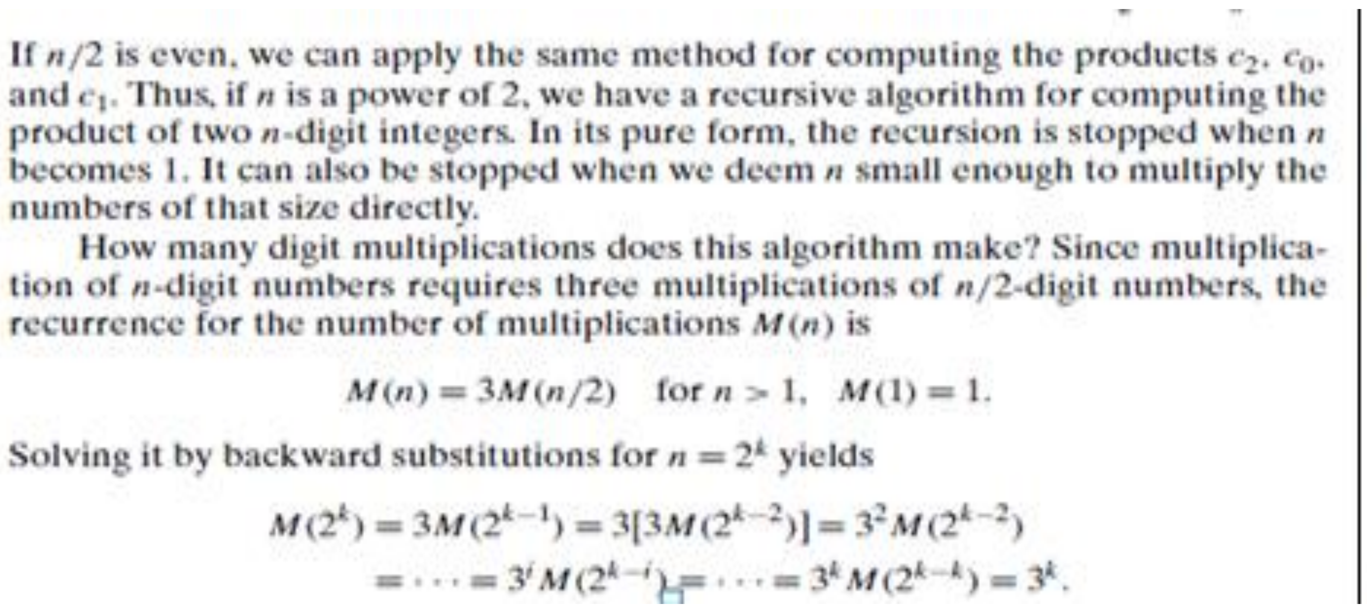
$$\begin{aligned} c = a * b &= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\ &= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0) \\ &= c_2 10^n + c_1 10^{n/2} + c_0, \end{aligned}$$

where

$c_2 = a_1 * b_1$ is the product of their first halves,

$c_0 = a_0 * b_0$ is the product of their second halves,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.



If $n/2$ is even, we can apply the same method for computing the products c_2, c_0, and c_1. Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit integers. In its pure form, the recursion is stopped when n becomes 1. It can also be stopped when we deem n small enough to multiply the numbers of that size directly.

How many digit multiplications does this algorithm make? Since multiplication of n-digit numbers requires three multiplications of $n/2$-digit numbers, the recurrence for the number of multiplications $M(n)$ is

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Solving it by backward substitutions for $n = 2^k$ yields

$$\begin{aligned} M(2^k) &= 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2 M(2^{k-2}) \\ &= \cdots = 3^i M(2^{k-i}) = \cdots = 3^k M(2^{k-k}) = 3^k. \end{aligned}$$

Example :

To demonstrate the basic idea of the algorithm, let us start with a case of

Four-digit integers –6721 and 3032 . These numbers can be represented as follows:

X= 3421 = 67 * 10^2 + 21 Let A = 67 ; B = 21

Y=3032 = 30 * 10^2 + 32 Let C = 30; D = 32

Now let us multiply them:

X* Y = AC * 10^4 + [AC + (A – B)* (D – C) + BD] * 10^2 + BD

 = 67 * 60 * 10^4 +[(67 * 60) +(67 – 21) * (32 -30) * 10^2 + (21 * 32)

 = 4020 * 10000 + [4020 +92 + 672] * 100 + 672

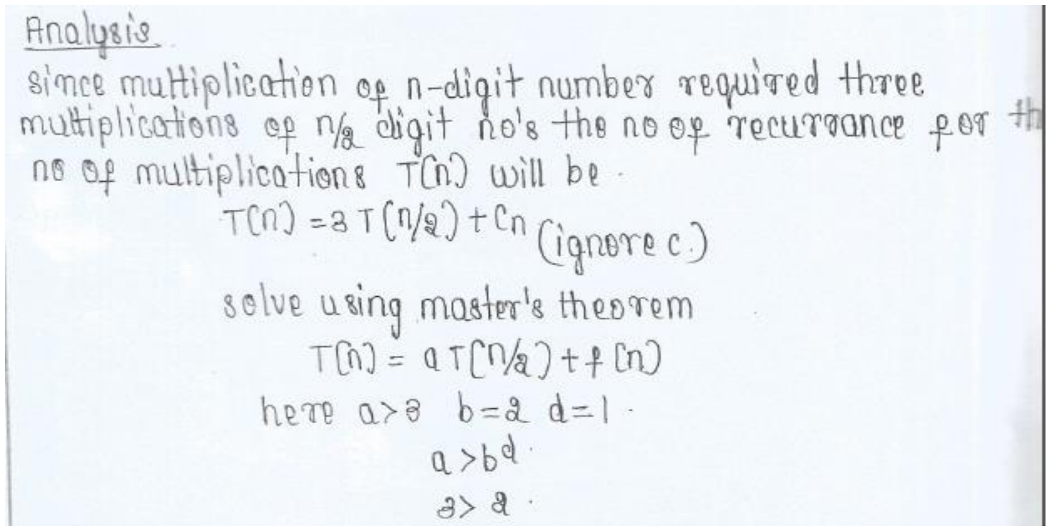
 = 40200000 + 478400 +672

 = 40679072

```
Algorithm multi( X, Y, n)
//Input : X & Y two long integers; n – no.of digits of X
// Output : Product of two long integers
Begin
   If ( n == 1)
```

```
        Return( X * Y)
    Else
        A= Left   n/2  bits of  X
        B = Right   n/2  bits of   X
        C= Left   n/2  bits of   Y
        D = Right   n/2  bits of   Y
        m1 = multi( A,C)
        m2 = multi( A-B, D-C)
        m3  = multi( B,D)
        return (m1 * $10^n$  +( m1 + m2  +m3) * $10^{n/2}$ +$m^3$
 End
```

Analysis



Analysis

since multiplication of n-digit number required three multiplications of n/2 digit no's the no of recurrance for th
no of multiplications T(n) will be.

$T(n) = 3T(n/2) + Cn$ (ignore c)

solve using master's theorem

$T(n) = aT(n/2) + f(n)$

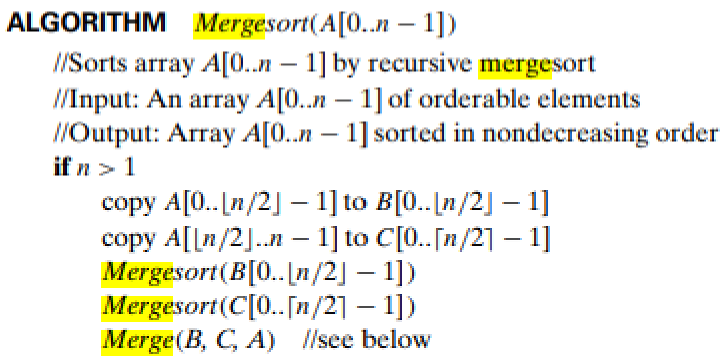
here $a>3$ $b=2$ $d=1$.

$a > b^d$.

$3 > 2$.

Hence T(n) = θ($n^{\log_2 3}$) = θ($n^{1.58}$). This time complexity is much better than the brute force multiplication which takes θ(n^2) time for n digit multiplication.

8.



```
ALGORITHM  Mergesort(A[0..n − 1])
    //Sorts array A[0..n − 1] by recursive mergesort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    if n > 1
        copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
        copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
        Mergesort(B[0..⌊n/2⌋ − 1])
        Mergesort(C[0..⌈n/2⌉ − 1])
        Merge(B, C, A)    //see below
```

Given that the `merge` function runs in \Theta(n)Θ(*n*)\Theta, left parenthesis, n, right parenthesis time when merging nnn elements, how do we get to showing that `mergeSort` runs in \Theta(n \log_2 n)Θ(*n*log2*n*)\Theta, left parenthesis, n, log, start base, 2, end base, n, right parenthesis time? We start by thinking about the three parts of divide-and-conquer and how to account for their running times. We assume that we're sorting a total of nnn elements in the entire array.

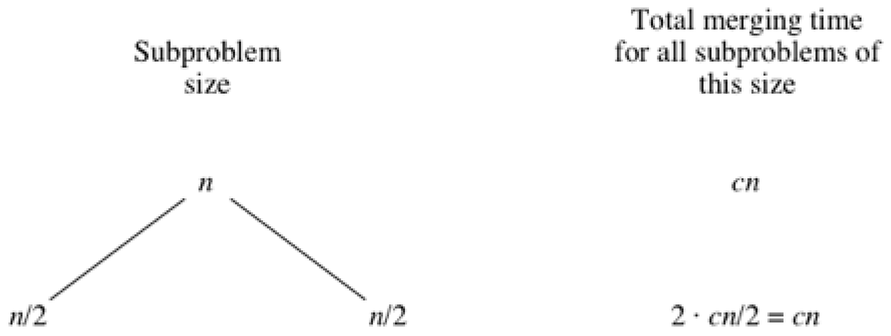
1. The divide step takes constant time, regardless of the subarray size. After all, the divide step just computes the midpoint qqq of the indices ppp and rrr. Recall that in big-Θ notation, we indicate constant time by \Theta(1)Θ(1)\Theta, left parenthesis, 1, right parenthesis.
2. The conquer step, where we recursively sort two subarrays of approximately n/2n/2n, slash, 2 elements each, takes some amount of time, but we'll account for that time when we consider the subproblems.
3. The combine step merges a total of nnn elements, taking \Theta(n)Θ(n)\Theta, left parenthesis, n, right parenthesis time.

If we think about the divide and combine steps together, the \Theta(1)Θ(1)\Theta, left parenthesis, 1, right parenthesis running time for the divide step is a low-order term when compared with the \Theta(n)Θ(n)\Theta, left parenthesis, n, right parenthesis running time of the combine step. So let's think of the divide and combine steps together as taking \Theta(n)Θ(n)\Theta, left parenthesis, n, right parenthesis time. To make things more concrete, let's say that the divide and combine steps together take cncnc, n time for some constant ccc.

To keep things reasonably simple, let's assume that if n>1n>1n, is greater than, 1, then nnn is always even, so that when we need to think about n/2n/2n, slash, 2, it's an integer. (Accounting for the case in which nnn is odd doesn't change the result in terms of big-Θ notation.) So now we can think of the running time of `mergeSort` on an nnn-element subarray as being the sum of twice the running time of `mergeSort` on an (n/2)(n/2)left parenthesis, n, slash, 2, right parenthesis-element subarray (for the conquer step) plus cncnc, n (for the divide and combine steps—really for just the merging).

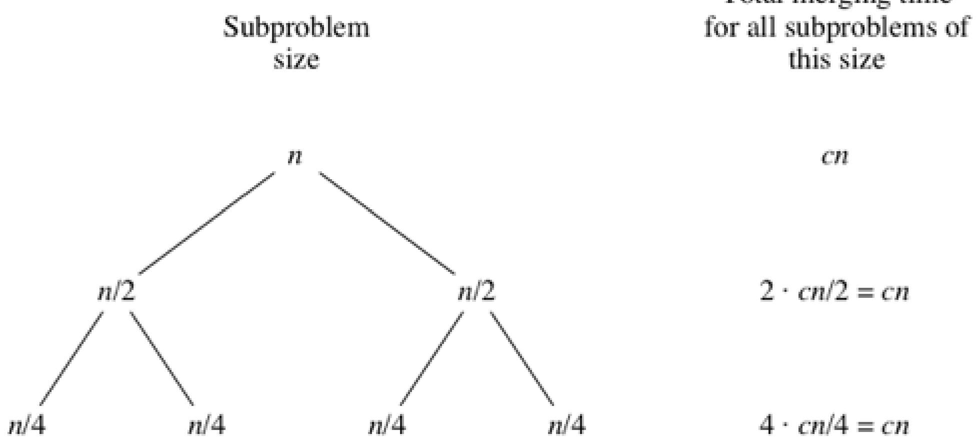
Now we have to figure out the running time of two recursive calls on n/2n/2n, slash, 2 elements. Each of these two recursive calls takes twice of the running time of `mergeSort` on an (n/4)(n/4)left parenthesis, n, slash, 4, right parenthesis-element subarray (because we have to halve n/2n/2n, slash, 2) plus cn/2cn/2c, n, slash, 2 to merge. We have two subproblems of size n/2n/2n, slash, 2, and each takes cn/2cn/2c, n, slash, 2 time to merge, and so the total time we spend merging for subproblems of size n/2n/2n, slash, 2 is 2\cdot cn/2 = cn2·cn/2=cn2, dot, c, n, slash, 2, equals, c, n.

Let's draw out the merging times in a "tree":



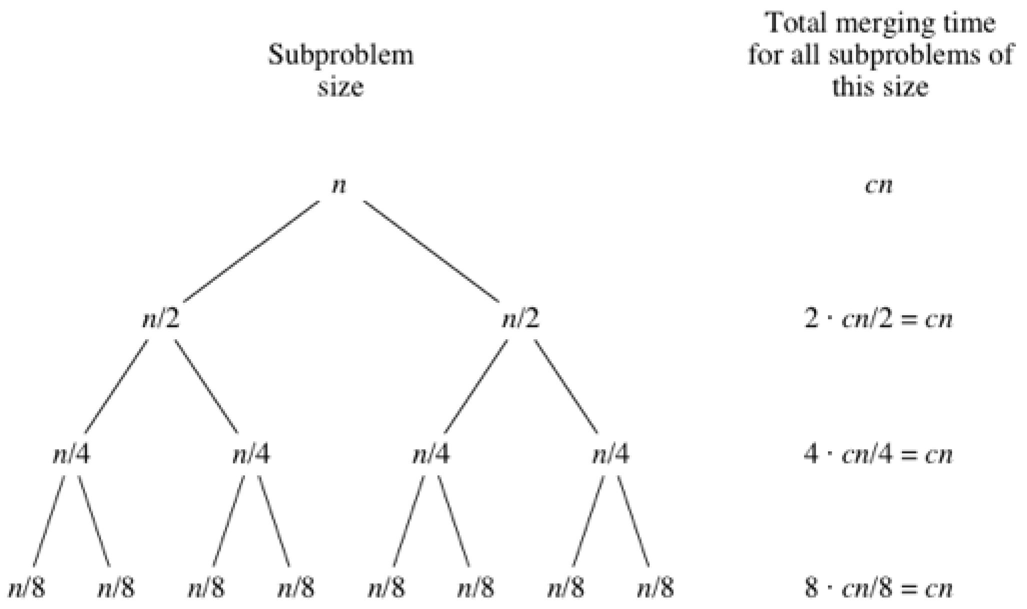
A diagram with a tree on the left and merging times on the right. The tree is labeled "Subproblem size" and the right is labeled "Total merging time for all subproblems of this size." The first level of the tree shows a single node n and corresponding merging time of c times n. The second level of the tree shows two nodes, each of 1/2 n, and a merging time of 2 times c times 1/2 n, the same as c times n.

Computer scientists draw trees upside-down from how actual trees grow.
A tree is a graph with no cycles (paths that start and end at the same place). Convention is to call the vertices in a tree its nodes. The root node is on top; here, the root is labeled with the n*n*n subarray size for the original n*n*n-element array. Below the root are two child nodes, each labeled n/2*n*/2n, slash, 2 to represent the subarray sizes for the two subproblems of size n/2*n*/2n, slash, 2.
Each of the subproblems of size n/2*n*/2n, slash, 2 recursively sorts two subarrays of size (n/2)/2(*n*/2)/2left parenthesis, n, slash, 2, right parenthesis, slash, 2, or n/4*n*/4n, slash, 4. Because there are two subproblems of size n/2*n*/2n, slash, 2, there are four subproblems of size n/4*n*/4n, slash, 4. Each of these four subproblems merges a total of n/4*n*/4n, slash, 4 elements, and so the merging time for each of the four subproblems is cn/4*cn*/4c, n, slash, 4. Summed up over the four subproblems, we see that the total merging time for all subproblems of size n/4*n*/4n, slash, 4 is 4 \cdot cn/4 = cn4·*cn*/4=*cn*4, dot, c, n, slash, 4, equals, c, n:



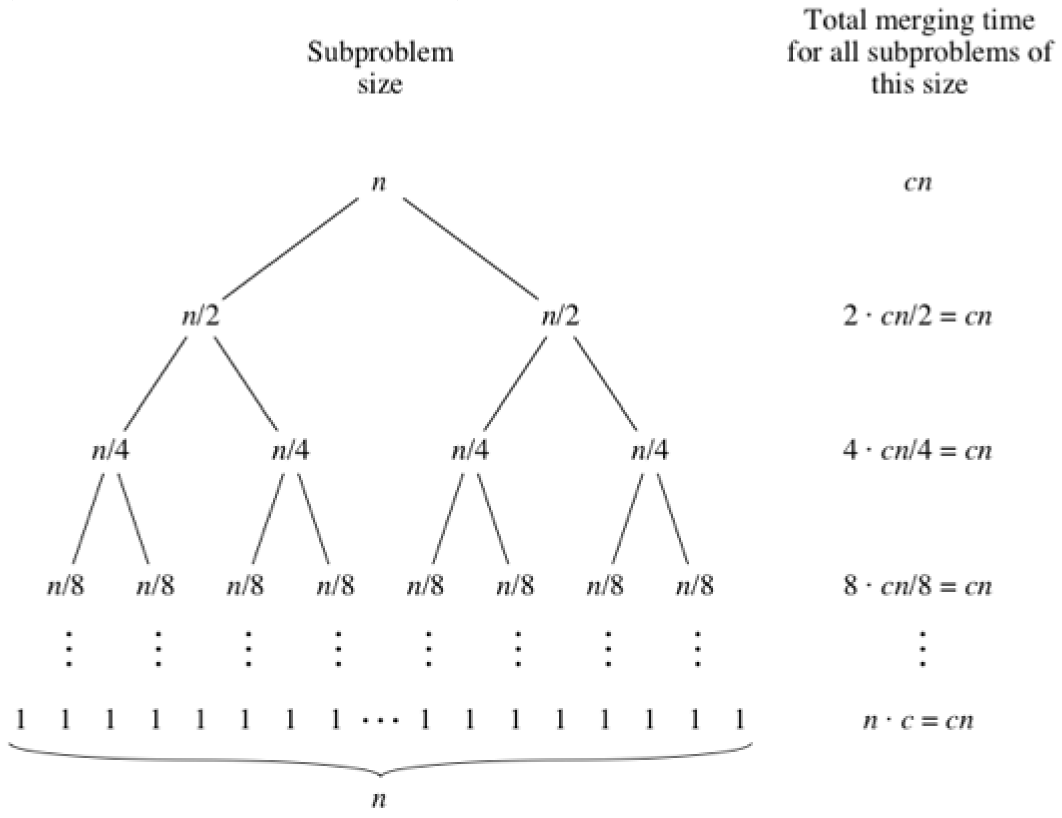
A diagram with a tree on the left and merging times on the right. The tree is labeled "Subproblem size" and the right is labeled "Total merging time for all subproblems of this size." The first level of the tree shows a single node n and corresponding merging time of c times n. The second level of the tree shows two nodes, each of 1/2 n, and a merging time of 2 times c times 1/2 n, the same as c times n. The third level of the tree shows four nodes, each of 1/4 n, and a merging time of 4 times c times 1/4 n, the same as c times n.

What do you think would happen for the subproblems of size n/8*n*/8n, slash, 8? There will be eight of them, and the merging time for each will be cn/8*cn*/8c, n, slash, 8, for a total merging time of 8 \cdot cn/8 = cn8·*cn*/8=*cn*8, dot, c, n, slash, 8, equals, c, n:



A diagram with a tree on the left and merging times on the right. The tree is labeled "Subproblem size" and the right is labeled "Total merging time for all subproblems of this size." The first level of the tree shows a single node n and corresponding merging time of c times n. The second level of the tree shows two nodes, each of 1/2 n, and a merging time of 2 times c times 1/2 n, the same as c times n. The third level of the tree shows four nodes, each of 1/4 n, and a merging time of 4 times c times 1/4 n, the same as c times n. The fourth level of the tree shows eight nodes, each of 1/8 n, and a merging time of 8 times c times 1/8 n, the same as c times n.

As the subproblems get smaller, the number of subproblems doubles at each "level" of the recursion, but the merging time halves. The doubling and halving cancel each other out, and so the total merging time is cn*cn*c, n at each level of recursion. Eventually, we get down to subproblems of size 1: the base case. We have to spend \Theta(1)Θ(1)\Theta, left parenthesis, 1, right parenthesis time to sort subarrays of size 1, because we have to test whether p < rp<rp, is less than, r, and this test takes time. How many subarrays of size 1 are there? Since we started with nnn elements, there must be nnn of them. Since each base case takes \Theta(1)Θ(1)\Theta, left parenthesis, 1, right parenthesis time, let's say that altogether, the base cases take cncnc, n time:



A diagram with a tree on the left and merging times on the right. The tree is labeled "Subproblem size" and the right is labeled "Total merging time for all subproblems of this size." The first level of the tree shows a single node n and corresponding merging time of c times n. The second level of the tree shows two nodes, each of 1/2 n, and a merging time of 2 times c times 1/2 n, the same as c times n. The third level of the tree shows four nodes, each of 1/4 n, and a merging time of 4 times c times 1/4 n, the same as c times n. The fourth level of the tree shows eight nodes, each of 1/8 n, and a merging time of 8 times c times 1/8 n, the same as c times n. Underneath that level, dots are shown to indicate the tree continues like that. A final level is shown with n nodes of 1, and a merging time of n times c, the same as c times n.

Now we know how long merging takes for each subproblem size. The total time for `mergeSort` is the sum of the merging times for all the levels. If there are lll levels in the tree, then the total merging time is l \cdot cnl·cnl, dot, c, n. So what is lll? We start with subproblems of size nnn and repeatedly halve until we get down to subproblems of size 1. We saw this characteristic when we analyzed

binary search, and the answer is l = \log_2 n + 1*l*=log2*n*+1l, equals, log, start base, 2, end base, n, plus, 1. For example, if n=8*n*=8n, equals, 8, then \log_2 n + 1 = 4log2 *n*+1=4log, start base, 2, end base, n, plus, 1, equals, 4, and sure enough, the tree has four levels: n = 8, 4, 2, 1*n*=8,4,2,1n, equals, 8, comma, 4, comma, 2, comma, 1. The total time for `mergeSort`, then, is cn (\log_2 n + 1)*cn*(log2*n*+1)c, n, left parenthesis, log, start base, 2, end base, n, plus, 1, right parenthesis. When we use big-Θ notation to describe this running time, we can discard the low-order term (+1+1plus, 1) and the constant coefficient (c*c*c), giving us a running time of \Theta(n \log_2 n)Θ(*n*log2*n*)\Theta, left parenthesis, n, log, start base, 2, end base, n, right parenthesis, as promised.
One other thing about merge sort is worth noting. During merging, it makes a copy of the entire array being sorted, with one half in `lowHalf` and the other half in `highHalf`. Because it copies more than a constant number of elements at some time, we say that merge sort does not work in place. By contrast, both selection sort and insertion sort do work in place, since they never make a copy of more than a constant number of array elements at any one time. Computer scientists like to consider whether an algorithm works in place, because there are some systems where space is at a premium, and thus in-place algorithms are preferred.